Mid term - Bi-grams

Author
Marten Schuitemaker
E-mail address
marten.schuitemaker@etu.univ-nantes.fr
Computer
Macbook air M1 (2020)

Abstract

This report presents a comparative analysis of sequential and parallel computation approaches for generating bi-grams from text corpora. Bi-grams, pairs of adjacent elements extracted from a sequence of tokens, play a crucial role in various natural language processing tasks. We implemented both sequential and parallel algorithms for bi-gram generation, conducted experiments on corpora of different sizes, and evaluated the performance in terms of execution time. The results demonstrate a significant improvement in performance with parallel computation, particularly for larger corpora. We discuss the scalability, efficiency, and trade-offs involved in parallelization, highlighting the benefits of leveraging multiple CPU cores for text processing tasks.

GitHub link : https://github.com/Martenren/mid-term-pp.
The computer used on has for specs :
8-core CPU with 4 performance cores and 4 efficiency cores
Max CPU clock rate 3.2 GHz

1. Introduction to bi-grams

2. Introduction to Bi-gram Generation

Bi-grams, also known as digrams, are pairs of adjacent elements extracted from a sequence of tokens. In the context of natural language processing (NLP), these tokens typically represent words or characters in a text. The process of bi-gram generation involves iterating through a corpus of text and extracting consecutive pairs to analyze the text's structure or to perform various computational linguistic tasks.

2.1. Definition and Concept

A bi-gram model views a text as a sequence of overlapping pairs, where each pair consists of two components: the previous token and the current token. Mathematically, if we have a sequence of tokens $T = t_1, t_2, ..., t_N$, the bi-grams would be the pairs $(t_1, t_2), (t_2, t_3), ..., (t_{N-1}, t_N)$.

2.2. Visual representation

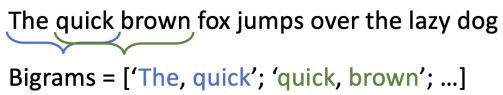

Figure 1. Example of bi-gram generation

2.3. Applications of Bi-gram Generation

Bi-gram generation is a fundamental technique used in various applications within NLP and computational linguistics, including but not limited to:

- **Language Modeling:** Bi-grams are used to build simple statistical language models that can predict the likelihood of a word given the previous word. These models are essential in applications like predictive text input, spelling correction, and speech recognition.

- **Text Analysis:** By analyzing the frequency and distribution of bi-grams within a text, researchers can gain insights into language patterns, stylistic features, and content structure.
- **Machine Translation:** Bi-gram models can assist in translating text from one language to another by analyzing word pair occurrences and their translations in parallel corpora.
- **Information Retrieval:** Search engines and document classification systems use bi-grams to improve the accuracy of keyword searches and to better understand document content for categorization.

In C++, bi-gram generation can be efficiently implemented using standard libraries to handle string manipulation and data storage. The following chapters will delve into the technical implementation details and the specific applications of bi-grams in our project.

3. Sequential Code Implementation

To implement the computation of bi-grams for a corpus of text, we utilize the C++ programming language, leveraging its Standard Template Library (STL) for efficient data manipulation. The code, which is available on GitHub, contains a class, `BigramGenerator`, that takes a text file as input and generates bi-grams for the words contained within it.

3.1. Function Overview

The `BigramGenerator` class performs the following steps:

1. Opens the text file and reads words into a vector `corpus`.
2. Defines a member function `generateBigrams` that computes the bi-grams from the `corpus`.
3. Within `generateBigrams`, initializes a vector of pairs to store the bi-grams.
4. Iterates over the `corpus` and constructs each bi-gram by pairing adjacent words.
5. Returns the vector of bi-grams.

3.2. Algorithm Description

The algorithm for generating bi-grams follows a straightforward approach:

1. Initialize an empty vector to hold the bi-grams.
2. Traverse the `corpus` vector using a loop.
3. For each word, create a pair with the subsequent word to form a bi-gram.
4. Add the bi-gram to the bi-grams vector.
5. Repeat steps 2–4 until the end of the `corpus` is reached, ensuring that the last word is not used to form an incomplete bi-gram.

3.3. Code Overview

The C++ code uses a class structure to encapsulate the functionality of bi-gram generation. The `generateBigrams` member function utilizes a single loop to traverse the `corpus` vector and uses the STL function `std::make_pair` to create each bi-gram.

3.4. Efficiency Considerations

The sequential implementation is straightforward but may not be the most efficient, especially for large corpora. The time complexity of the bi-gram generation is $O(n)$, where n is the number of words in the corpus. For very large datasets, the memory overhead of storing all bi-grams and the time spent on string manipulation could become significant.

3.5. Performance Measurement

Performance is measured by timing the bi-gram generation process for corpora of varying sizes. The durations are recorded in a vector `sequential_times` with each entry corresponding to the time taken to generate bi-grams for a specific corpus size.

3.6. Conclusion

The sequential code implementation provides a correct and functional method to generate bi-grams from a text corpus using C++. While suitable for smaller datasets, its performance for larger corpora highlights the need for potential optimization techniques such as parallel processing or more advanced data structures to improve computational efficiency.

4. Parallel Code Implementation

The limitations of sequential bi-gram generation become more pronounced with the increase in corpus size. To address these limitations, we introduce a parallelized approach that utilizes the OpenMP library to distribute the work across multiple threads, thereby accelerating the process on multicore processors.

4.1. Parallel Computing Overview

Parallel computing involves the simultaneous use of multiple compute resources to solve a computational problem. This is achieved by breaking down the problem into independent tasks that can be executed concurrently. In the context of bi-gram generation, the corpus can be divided into segments, with each thread processing a different segment to generate bi-grams.

4.2. Implementation with OpenMP

OpenMP (Open Multi-Processing) is a widely-used, portable, and scalable model that provides a simple and flexible interface for developing parallel applications in C, C++, and Fortran on shared memory architectures. Our parallel code implementation employs OpenMP directives to create a parallel region and distribute the bi-gram generation workload across the available threads.

4.3. Function Overview

The `generateBigramsParallel` member function of the `BigramGenerator` class performs the parallel bi-gram generation:

1. Receives the number of threads to be used as a parameter.
2. Creates a vector of pairs to store the bi-grams, just as in the sequential version.
3. Defines a parallel region with OpenMP directives specifying the number of threads.
4. Distributes the iterations of the loop across the threads using the `pragma omp for` directive.
5. Each thread independently computes its assigned portion of bi-grams.
6. Combines the results from all threads and returns the complete vector of bi-grams.

4.4. Algorithm Description

The parallel algorithm follows a similar structure to the sequential algorithm with key differences in execution:

1. Initialize an empty vector to hold the bi-grams with reserved space for all bi-gram pairs.
2. Use OpenMP to create a parallel region, specifying the number of threads for execution.
3. The loop is distributed among the threads, with each thread processing a different segment of the `corpus`.
4. Within each thread, construct bi-grams for the assigned segment and store them in the shared vector.
5. Upon completion, merge the results into a single vector containing all bi-grams.

4.5. Code Overview

The parallel version of the code uses the OpenMP directives to manage the parallel region and work distribution. The `pragma omp parallel` directive is used to begin a parallel region, and the `pragma omp for` directive is used to distribute the loop iterations among the threads.

4.6. Efficiency Considerations

The parallel implementation aims to reduce the overall execution time by utilizing multiple cores available on modern CPUs. It is important to consider factors such as thread overhead, synchronization costs, and the division of work among threads to achieve optimal performance.

4.7. Performance Measurement

Performance is evaluated by measuring the time taken to generate bi-grams in parallel for various thread counts. These measurements are stored in a vector `parallel_times_threads` for further analysis. We expect a reduction in execution time with an increase in the number of threads, up to a certain point, beyond which performance may plateau or even degrade due to overheads.

4.8. Conclusion

Parallel processing presents an effective strategy for accelerating bi-gram generation in large text corpora. By leveraging the computational power of multicore CPUs and optimizing the division of labor among threads, we can significantly improve the performance of our bi-gram generation tool. The parallel implementation not only demonstrates the capability of modern computing systems but also serves as a practical example of applying parallel computing techniques to solve real-world problems in natural language processing.

5. Experimental Results and Discussion

In the subsequent chapters, we will present a detailed analysis of the experimental results obtained from both the sequential and parallel implementations. We will discuss the impact of varying corpus sizes, the number of threads, and the trade-offs involved in parallelization. The findings will illustrate the scalability and efficiency of the parallel approach in comparison to the sequential implementation.

6. Results: Sequential vs. Parallel Computation

In this section, we present the results of comparing the performance of sequential and parallel computation for generating bi-grams from corpora of different sizes. We evaluate the execution time for both approaches and analyze the impact of varying corpus sizes and thread counts on performance.

6.1. Experimental Setup

We conducted experiments using multiple corpora of different sizes, ranging from 10,000 to 100,000,000 words. The parallel implementation was tested with a fixed number of threads (8) to evaluate its scalability and efficiency.

6.2. Performance Comparison

The execution times for generating bi-grams using both sequential and parallel computation are summarized below:

Words	S (s)	P (s)
10,000	0.007	0.007
100,000	0.068	0.003
1,000,000	0.331	0.031
5,000,000	0.708	0.136
10,000,000	1.904	0.296
25,000,000	4.339	0.741
50,000,000	6.011	1.676
75,000,000	8.995	2.521
100,000,000	9.585	4.821

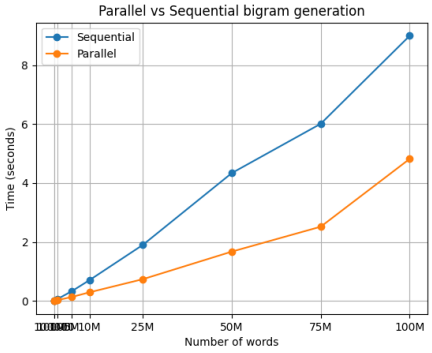

Figure 2. Sequential vs Parallel on different size corpora

Additionally, we tested the parallel implementation with varying numbers of threads on a single corpus of 100,000,000 words:

Threads	Parallel time (s)
1	9.585
2	5.831
3	4.824
4	3.993
5	3.804
6	3.920
7	3.753
8	3.512
9	3.376
10	3.470
11	3.272
12	3.216

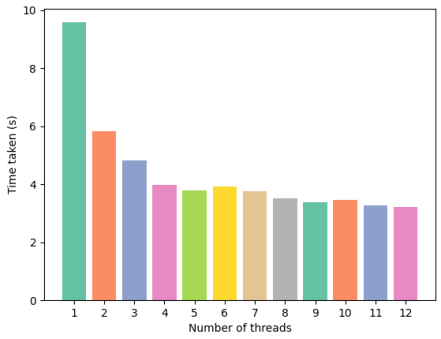

Figure 3. Execution time for a 100M corpus per thread count

6.3. Discussion

The results demonstrate a significant performance improvement with parallel computation compared to the sequential approach. On average, the parallel implementation achieves a speedup of approximately two times over the sequential method across higher corpuss sizes.

Furthermore, as the corpus size increases, the speedup provided by parallel computation becomes more pronounced. This is evident from the decreasing execution times relative to the sequential approach for larger corpora.

Regarding the scalability of the parallel implementation with varying thread counts, we observe diminishing returns beyond a certain number of threads. While increasing the number of threads initially leads to improved performance, the benefits plateau or even decrease due to overheads associated with thread management and synchronization.

Overall, these results highlight the effectiveness of parallel computation for bi-gram generation, particularly for large-scale text processing tasks. By leveraging multiple CPU cores, parallelization enables significant reductions in execution time, thereby enhancing the efficiency of natural language processing workflows.

7. Conclusion

In conclusion, our study demonstrates the effectiveness of parallel computation in accelerating bi-gram generation from text corpora. By distributing the workload across multiple threads, we achieve significant reductions in execution time, particularly for large-scale datasets. However, we also observe diminishing returns beyond a certain number of threads due to overheads associated with thread management and synchronization. Nevertheless, parallel computation presents a practical solution for enhancing the efficiency of natural language processing workflows.